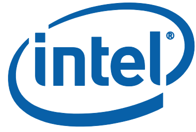

White Paper

A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

Star Zeng
Intel Corporation

September 2015

Executive Summary

This paper presents the internal structure and boot flow of the SMM-based UEFI Authenticated Variable driver in the MDE Module Package and Security Package of the EDKII.

Prerequisite
This paper assumes that audience has EDKII/UEFI firmware development experience. He or she should also be familiar with UEFI/PI firmware infrastructure, such as SEC, PEI, DXE, runtime phase.

Table of Contents

Overview 5

Introduction to the SMM Authenticated Variable 5

Threat Model 6

Part I - SMM 8

Why SMM? 8

SMM communication 8

Part II - Authentication 10

Why Authentication? 10

Authenticated Variable Input Format 10

Authentication Flow 11

Concept: SETUP MODE V.S USER MODE 13

Concept: STANDARD MODE V.S. CUSTOM MODE 15

Concept: SecureBoot ENABLE V.S. DISABLE 15

Concept: UserPhysicalPresent 16

Authentication Flow - ProcessVarWithPk 16

Authentication Flow - ProcessVarWithKek 17

Authentication Flow - ProcessVariable 18

Part III - Variable 19

Variable Storage Format 19

Variable Update Flow 20

Variable Reclaim 20

Fault Tolerant Write 21

Part IV – Robust Consideration 25

Variable Lock 25

Variable Check 26

Variable Check – HII Based Checker 28

Variable Check – PCD Based Checker 30

Variable Check – Variable Property 31

Variable Quota Management 32

Variable Quota Management – Allocation 32
Variable Quota Management – Error Logging 34
Variable Quota Management – Recovery 34
Conclusion 35
Glossary 36
References 37

Overview

Introduction to the SMM Authenticated Variable

UEFI Authenticated Variable is designed to provision and maintain the UEFI secure boot status. The platform firmware keys, like 1) Platform Keys, 2) Key Exchange Keys, 3) Image Signature Database, are all stored in UEFI authenticated variable. Various sets of documentation can be found on UEFI Secure boot and securing the platform [SECURE1][SECURE2][SECURE3].

EDKII is open source implementation for UEFI firmware. In MdeModulePkg, Universal\Variable\RuntimeDxe is an implementation of UEFI variable driver. This driver links AuthVariableLib, which can provide UEFI variable authentication services. In SecurityPkg, Library\AuthVariableLib is an implementation of authenticated services defined in UEFI specification. If a platform does not want UEFI authenticated services (for example, NT32 emulation), it can just link a NULL implementation in MdeModulePkg, Library\AuthVariableLibNull. This document will introduce detail on how EDKII SMM Authenticated Variable driver works.

Today UEFI Authenticated variables are managed by the IA firmware.

UEFI Authenticated Variables are a type of UEFI variable that includes the "Authenticated Write" attribute **EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS.** When this bit is set, these variables need to be cryptographically verified for updates. The SetVariable() API is in the UEFI Specification, chapter 7.2 [UEFI]. This is an API exposed by the IA firmware.

Today to protect the UEFI authenticated variables, the IA firmware will generate a System Manage Interrupt and pass control to System Management Mode (SMM). In SMM, there is a UEFI Variable driver with sources at https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Variable/RuntimeDxe including files https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/Variable/RuntimeDxe/VariableSmm.c and https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/Variable/RuntimeDxe/VariableSmmRuntimeDxe.c. VariableSmmRuntimeDxe.c is UEFI Runtime phase driver. It passes the variable access request to VariableSmm.c by using Software System Management Interrupt (SW SMI). The variableSmm.c exists in system management ram (SMRAM). The UEFI PI specification defines a software model for running code in SMM in volume 4 of the UEFI Platform Initialization (PI) specification [UEFI PI Specification].

This driver runs in SMM and expects to have another SMM driver, such as the closed sourced SMM FVB (Firmware Volume Block) driver, in order to write to SPI NOR from SMRAM. The PCH or other IOH is programmed such that the write access to SPI NOR can only occur from SMM.

Anyone can read the UEFI authenticated variables, but only the holder of a private key can modify the variables. The UEFI authentication support library (https://github.com/tianocore/edk2/blob/master/SecurityPkg/Library/AuthVariableLib/AuthService.c) uses a public key in a stored authenticated variable to guarantee that the private key holder truly did the variable update.

Details on Authenticated Variables can be found at 7.2.1 of the UEFI Specification.

SO today's art includes SMM for isolated execution and volatile storage, with SMM-based access control for writes to the SPI NOR.

OS ring 0 calls UEFI runtime service set variable, and the UEFI runtime firmware implementation of set variable generates an SMI and control passes to SMM. The UEFI SMM variable driver controls SPI NOR flash to manage the authenticated variables

Most OS's use authenticated variables with *AuthInfo.CertType* set to **EFI_CERT_TYPE_PKCS7_GUID**. As such, the code to support authenticated variables has to support PKCS7-based signatures, including X509 certificates, and cryptographic algorithms of SHA256 and RSA2048.

The EDK2 open source implementation exposes those primitives via the https://github.com/tianocore/edk2/tree/master/CryptoPkg. The CryptoPkg has the subset of API's required by the SMM-based UEFI Authenticated Variable driver. Underneath this API we use OpenSSL-based code.

For size of the authenticated variable store, Windows8.1 recommends http://msdn.microsoft.com/en-us/library/windows/hardware/dn423132.aspx
Reserved Memory for Windows Secure Boot UEFI Variables. A total of at least 64 KB of non-volatile NVRAM storage memory must be reserved for NV UEFI variables (authenticated and unauthenticated, BS and RT) used by UEFI Secure Boot and Windows, and the maximum supported variable size must be at least 32kB. There is no maximum NVRAM storage limit.

In the authenticated variable store we keep several special variables, namely the PK, KEK, db, dbx, dbt and dbr. (see 30.3~30.6 of the UEFI 2.5 specification).

Threat Model

The platform builder needs to avoid bypass of the logic that writes to the authenticated variable writes.

Physical attacks that replace SPI NOR or use Dediprog to modify the UEFI variables can roll-back or add errant entries. Similarly, software attacking where ring0 non-UEFI code can directly modify the flash region containing the authenticated variables poses another concern.

So the code that controls writes to the flash for the authenticated variables AND the code that verifies the signature in the AuthInfo of the authenticated variable MUST be protected at runtime from ring0 attacks (and provenance of this code must be guaranteed by some secure boot/update process).

From the above you can observe the need for an isolated execution (IsoW) to write to flash store of authenticated variables and verify (IsoV) signature of the authenticated variable when updating these variables.

Many platforms today use SMM and SPI NOR write trapping to have isolated, access controlled usage of this persistent store.

Summary

This section provided an overview of UEFI Authenticated Variable and EDKII. In next 4 sections, we will introduce 4 parts of SMM authenticated variable in EDKII. Part I focus on SMM, part II focus on authentication, part III focus on variable, part IV focus on robust consideration.

Part I - SMM

Why SMM?

An authenticated variable implementation requires an isolated execution environment to do the authentication and update flash. If there is no isolated environment, malicious code can bypass authentication check and update variable region directly.

System Management Mode (SMM) is an isolated environment defined in IA CPU architecture. SMM firmware infrastructure is defined in UEFI PI specification volume 4, and SMM core is implemented in EDKII.

SMM communication

SMM Authenticated Variable uses Smm Communication mechanism to pass parameter input by Variable service caller. During boot time, UEFI Variable driver will allocate a runtime communication buffer to use this buffer for parameter passing.

During runtime, when OS consumer calls variable services like GetVariable(), SetVariable(), the UEFI variable driver will copy data to communication buffer (copy the whole data, not the data pointer), then trigger Software SMI. In software SMI handler, SMM core infrastructure checks the “Header GUID” in SMM communication header, then dispatch to SMM Variable Handler. Then SMM Variable Handler checks the “Function ID” in Variable Communication Header, then dispatch to each variable services, like SmmGetVariable(), SmmSetVariable(). Then the SMM Variable services can get parameter, like GUID, Name, Data field, from VarirableAccess Communication Data.

Finally, SmmSetVariable() API can call the crypto services to do the authentication and update flash.

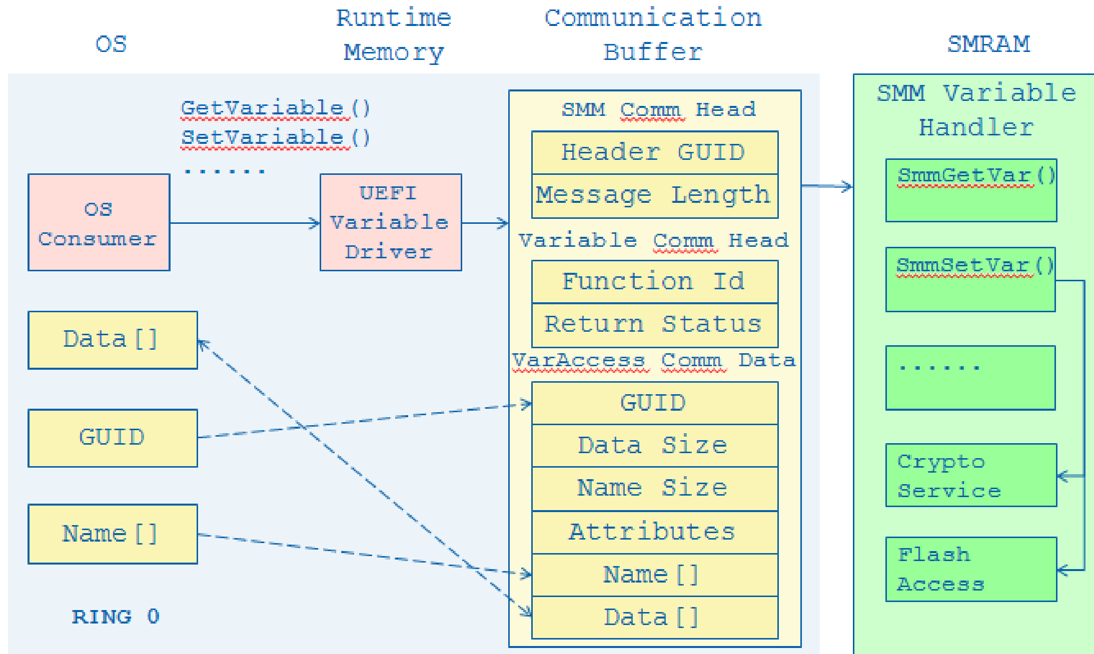

Figure 1 SMM Variable

Summary

This section describes the SMM part of SMM Authenticated Variable driver in EDKII, including why use SMM, and SMM communication.

Part II - Authentication

Why Authentication?

Authenticated Variable is designed to make sure the one who wants to update the variable is the one who has right to update the variable. If there is no authenticated, any malicious code can update a variable, which may break the system integrity and cause deny of service.

For a system with UEFI secure boot enabled, it need enroll 1) platform key, 2) key exchange keys, 3) image signature database. These keys are stored in variable, and the variable must be authenticated. Or the malicious code can break UEFI secure boot easily, by update the signature database.

Authenticated Variable Input Format

There are 2 kinds of authenticated variable.

1) Count based authenticated variable, if EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS attribute is set.
2) Time based authenticated variable, if EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS attribute is set. (Recommended)

When user calls SetVariable() API, the time based authenticated variable input data format is below.

There will be a time stamp associated with the authentication descriptor. Then the certificate type field, only PKCS7 is accepted for a time based authenticated variable. The certificate is DER-encoded PKCS #7 version 1.5 SignedData. The most important fields are 1) signer's DER-encoded X.509 certificate, 2) SHA256 hash of (VariableName, VariableGuid, Attributes, TimeStamp, New Variable Data). The Authenticated Variable driver will check the before update the variable content.

For count based authentication variable, the certificate type should be RSA_2048_SHA256. The certificate is in RSASSA_PKCS1-v1_5 format.

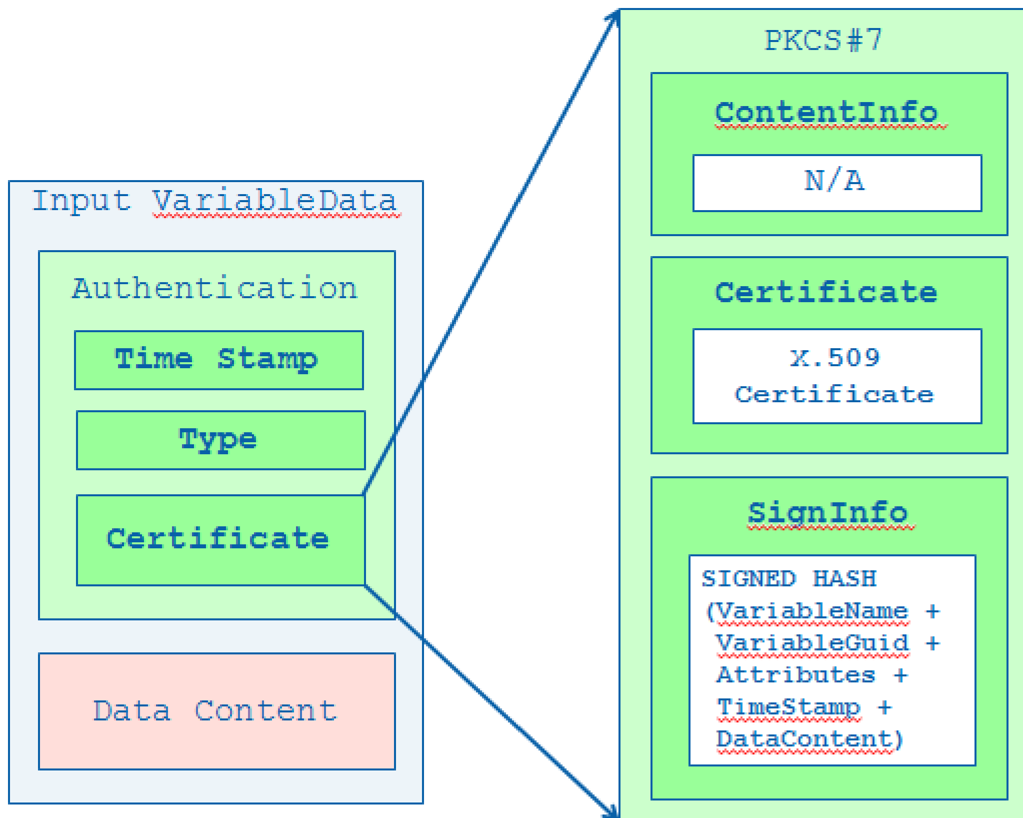

Figure 2: Authenticated Variable Input Format (Time based)

Authentication Flow

Authentication is needed when the consumer calls SetVariable() API. In a real variable driver (https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Universal/Variable/RuntimeDxe), the SetVariable() API first checks if authentication is supported. If authentication is supported, SetVariable() calls AuthVariableLibProcessVariable(). Or SetVariable() calls UpdateVariable() directly without authentication. The authentication services are provided as library (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Library/AuthVariableLib.h). The implementation of authentication library is at https://github.com/tianocore/edk2/tree/master/SecurityPkg/Library/AuthVariableLib.

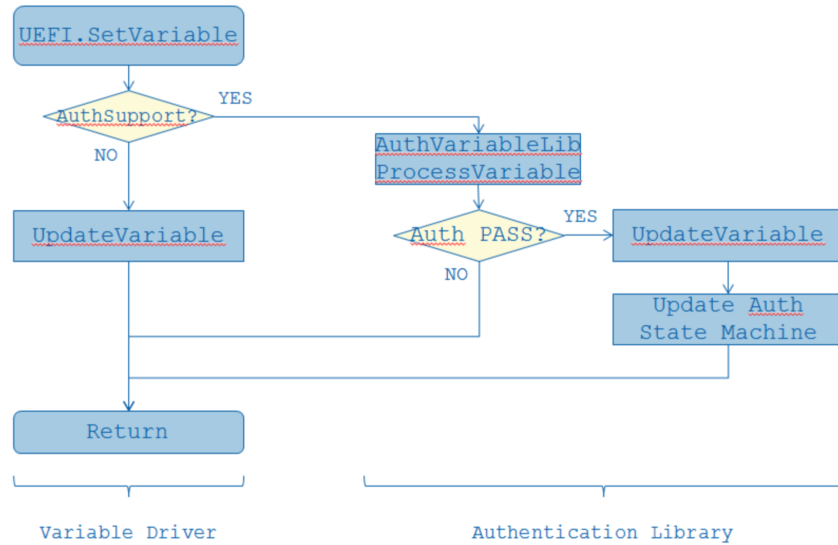

Figure 3: Variable Driver Flow - Overview

AuthVariableLibProcessVariable() checks if the variable is PK, or KEK, or Image Signature Database (db/dbx), or Authorized Timestamp Database (dbt) to do different action.

Platform Key (PK) is the key to establish a trust relationship between the platform owner and the platform firmware. The platform owner enrolls the public half of the key (PKpub) into the platform firmware. The platform owner can later use the private half of the key (PKpriv) to change platform ownership or to enroll a Key Exchange Key.

Key Exchange Key (KEK) is the key to establish a trust relationship between the operating system and the platform firmware. Each operating system (and potentially, each 3rd party application which need to communicate with platform firmware) enrolls a public key (KEKpub) into the platform firmware.

Image Signature Database (db/dbx) is the keys to maintain a list of authorized UEFI image (by db) and forbidden UEFI image (by dbx). The signature database is checked when UEFI boot manager is about to start a UEFI image. If UEFI image's signature is found in forbidden database, or not in authorized database, the UEFI image will be deferred and information placed in the Image Execution Information Table.

Authorized Timestamp Database (dbt). If a firmware supports the EFI_CERT_X509_SHA*_GUID signature types, it should support the RFC3161 timestamp specification. Images whose signature matches one of these types in the Forbidden Signature Database (dbx) shall only be considered forbidden if:

- the firmware either does not support timestamp verification, or
- the signature type has a time of revocation equal to zero, or
- the timestamp does not pass verification against the authorized timestamp and forbidden signature databases, or

- Finally the signature type's time of revocation is less than or equal to the time recorded in the image signature's timestamp.

If the timestamp's signature is authorized by the Authorized Timestamp Database (dbt) and the time recorded in the timestamp is less than the time of revocation, the image shall not be considered forbidden provided it is not forbidden by any other entry in the Forbidden Signature Database (dbx).

In general, below rules are applied during authenticated variable update:
1) PK variable update need be signed by old PK in USER MODE.
2) KEK variable update need be signed by PK in USER MODE.
3) Image Signature Database (db/dbx) or Authorized Timestamp Database (dbt) update need be signed by PK or KEK in USER MODE.
4) Other authenticated variable update need be signed by creator of this authenticated variable.

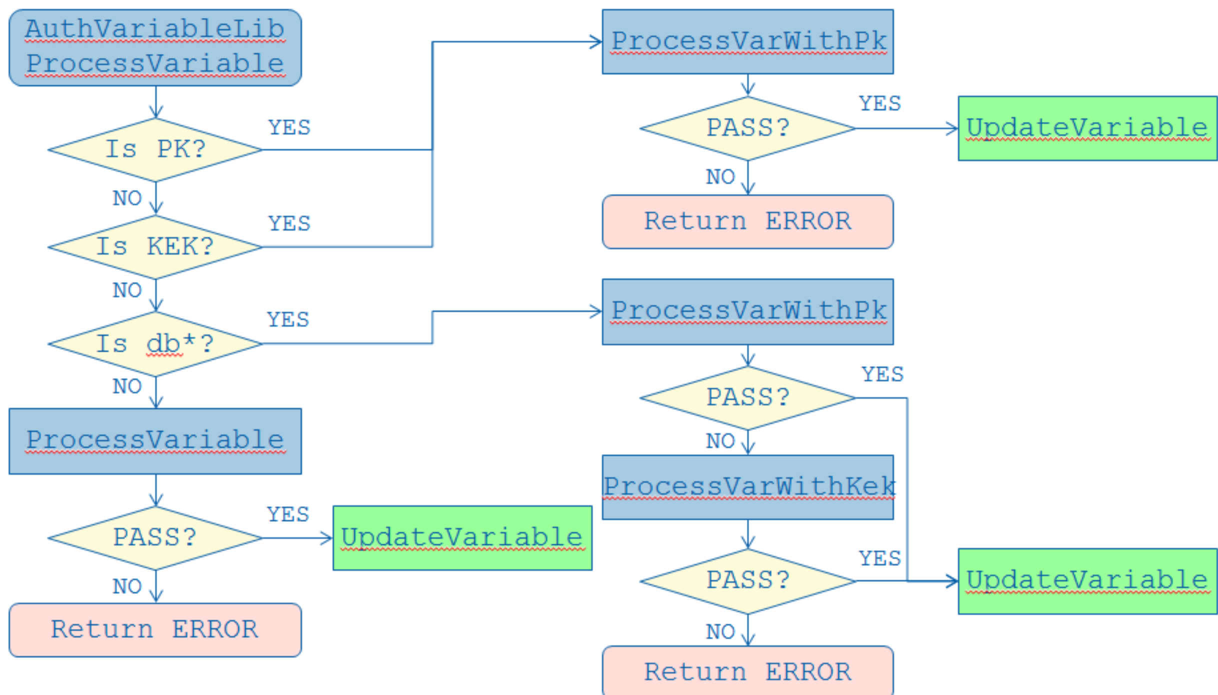

Figure 4: Variable Authentication Flow - Overview

Before we discuss detail authentication, let’s introduce some important concept at first. They will be used later.

Concept: SETUP MODE V.S USER MODE

According to UEFI specification, the system is in SETUP MODE when PK is NOT enrolled, while the system is in USER MODE when PK is enrolled.
This platform mode is shown as UEFI specification defined L”SetupMode” variable.

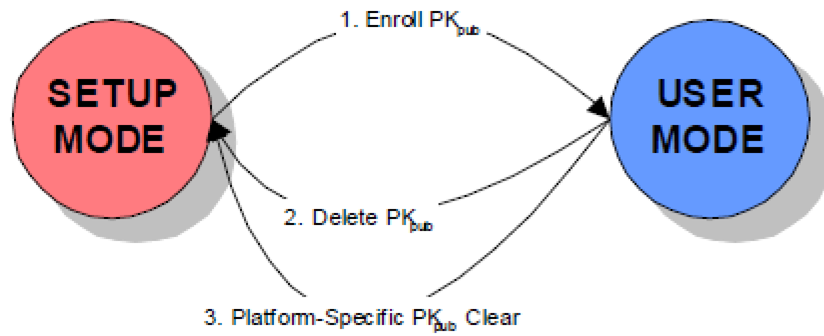

Figure 5: Setup Mode V.S User Mode

In UEFI 2.5 specification, 2 additional modes are added – DEPLOYED MODE and AUDIT MODE.

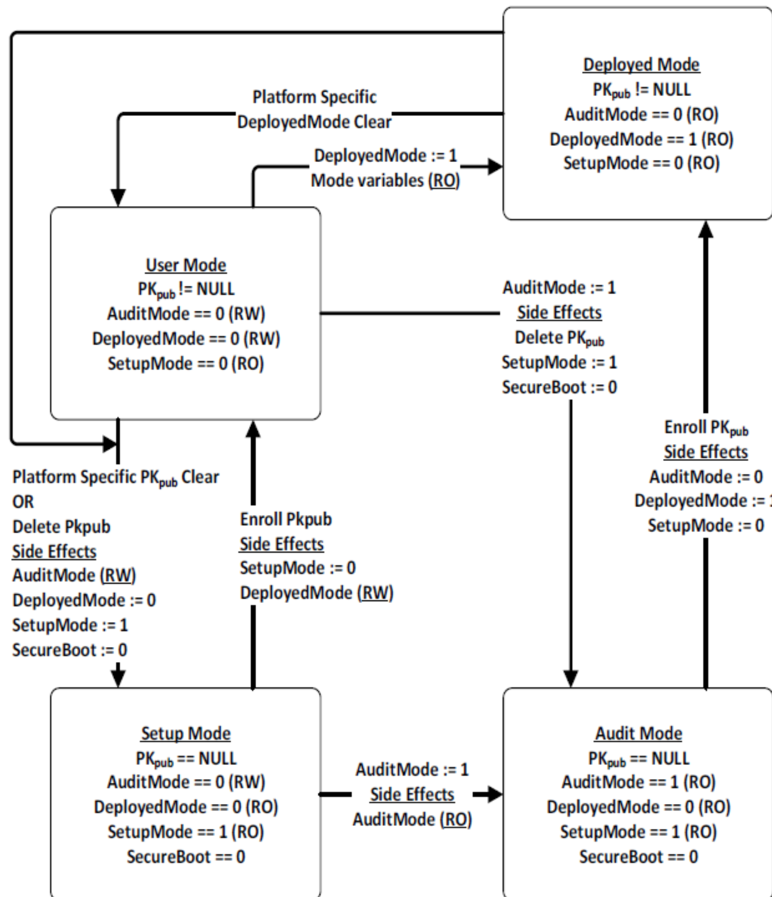

Figure 6: UEFI2.5 Secure Boot Modes

AUDIT MODE is extension for SETUP MODE. Audit Mode enables programmatic discovery of signature list combinations that successfully authenticate installed EFI images without the risk of rendering a system unbootable. Chosen signature lists configurations can be tested to ensure the system will continue to boot after the system is transitioned out of Audit Mode. After transitioning to Audit Mode, signature enforcement is disabled such that all images are initialized

and enhanced Image Execution Information Table (IEIT) logging is performed including recursive validation for multi-signed images.

DEPLOYED MODE is extension for USER MODE. Deployed Mode is the most secure mode. By design, both User Mode and Audit Mode support unauthenticated transitions to Deployed Mode. However, to move from Deployed Mode to any other mode requires a secure platform-specific method, or deleting the PK, which is authenticated.

Concept: STANDARD MODE V.S. CUSTOM MODE

This is EDKII implementation specific mode for UEFI secure boot. Standard Secure Boot mode is the default mode as UEFI Spec's description. Custom Secure Boot mode allows for more flexibility as specified in the following:

1) PK variable update need NOT be signed by old PK.
2) KEK variable update need NOT be signed by PK.
3) Image signature database (db/dbx) or Authorized Timestamp Database (dbt) update need NOT be signed by PK or KEK.

The switch between Standard Mode and Custom Mode need User Physical Present. It means a platform a way to detect a physical user present to do such action.
This mode is shown as EDKII implementation defined L"CustomMode" variable.

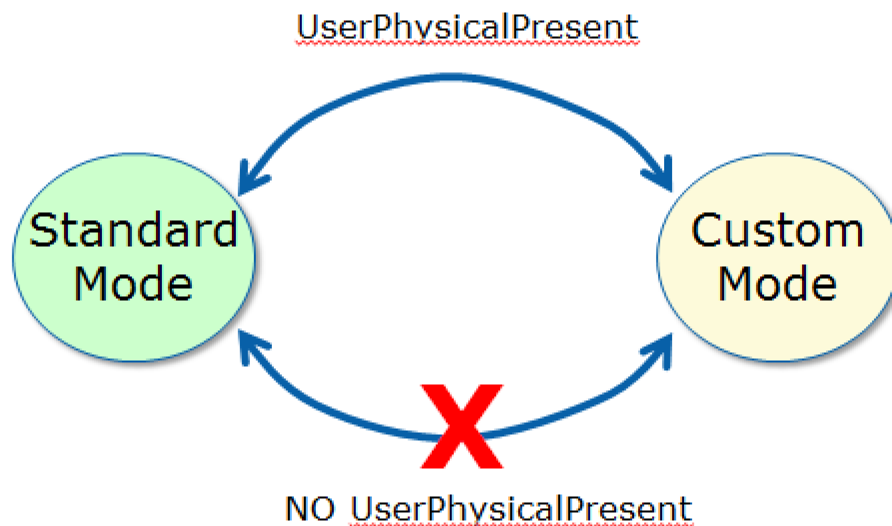

Figure 7: Standard Mode V.S Custom Mode

Concept: SecureBoot ENABLE V.S. DISABLE

In EDKII implementation, a platform can choose to enable or disable secure boot, if and only if user physical present. When SecureBoot mode is changed, it does not impact current boot. The new secure boot mode is adopted in next system boot.

This current secure boot state is shown as UEFI specification defined L"SetupBoot " variable. The new boot secure boot state is shown as EDKII implementation defined L"SecureBootEnable" variable.

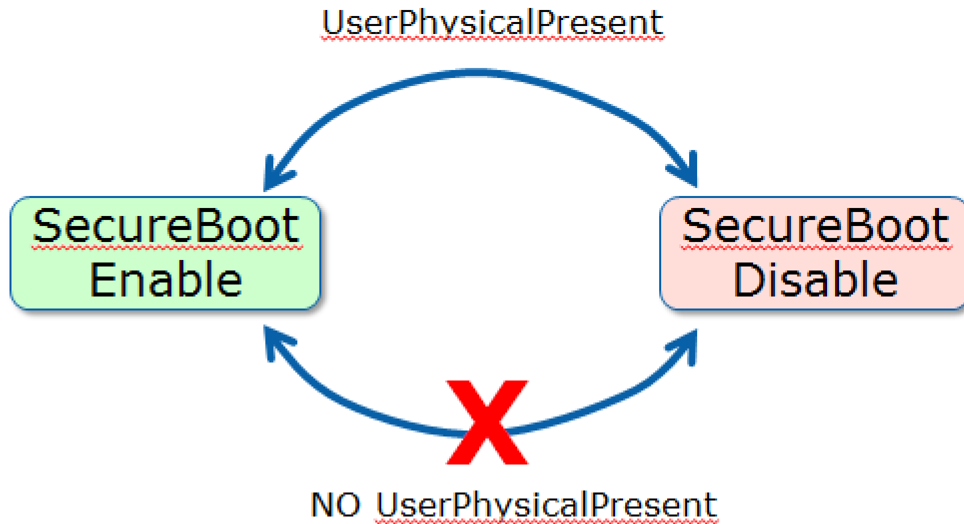

Figure 8: SecureBoot Enable V.S Disable

Concept: UserPhysicalPresent

In EDKII implementation, some secure boot variables change need user physical present to avoid remote attack. A platform need provide a library – PlatformSecureLib, which has UserPhysicalPresent() API. How to provide this API is platform implementation specific, and not addressed in this white paper

Now, we finish introduce the important concept, let's continue on detail of authentication flow.

Authentication Flow - ProcessVarWithPk

If in order to update PK or KEK, below authentication flow is used. First, if system is I CustomMode or UserPhysicalPresent, no authentication is needed. Second, if system is in SETUP mode not to enroll PK, no authentication is needed. Third, if system in USER mode, authentication with PK is performed. Last, if system in SETUP mode to enroll PK, authentication with this payload is performed.

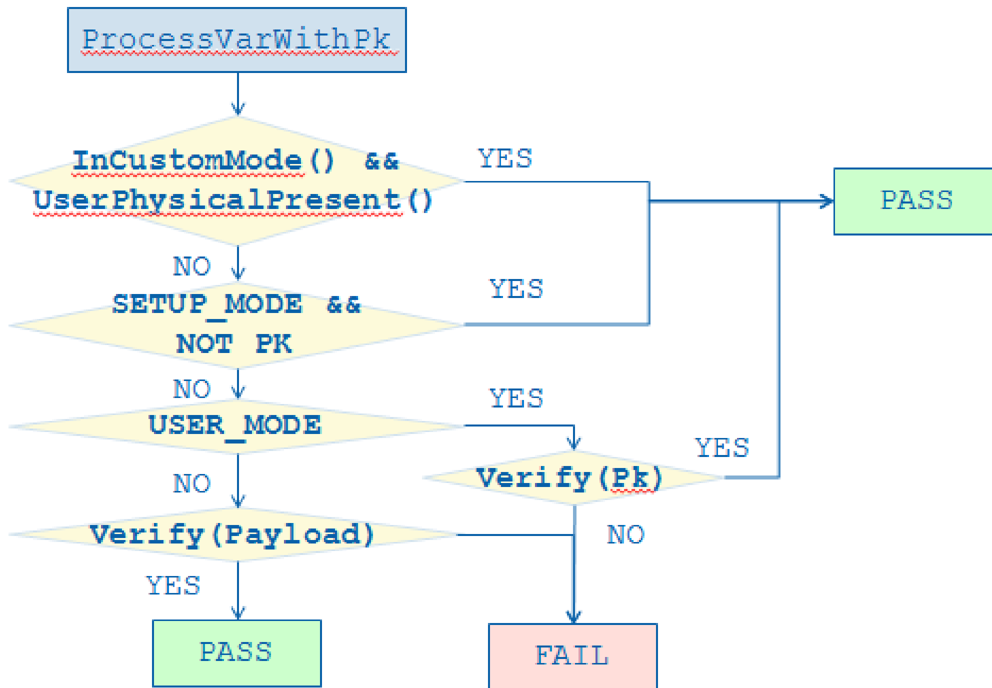

Figure 9: Variable Authentication Flow - ProcessVarWithPk

Authentication Flow - ProcessVarWithKek

If in order to update Image Signature Database (db/dbx) or Authorized Timestamp Database (dbt), system will invoke ProcessVarWithPk at firm. If fail, system will invoke ProcessVarWithKek. First, if system is CustomMode or UserPhysicalPresent, no authentication is needed. Second, if system is in SETUP mode, no authentication is needed. Last, authentication with KEK is performed.

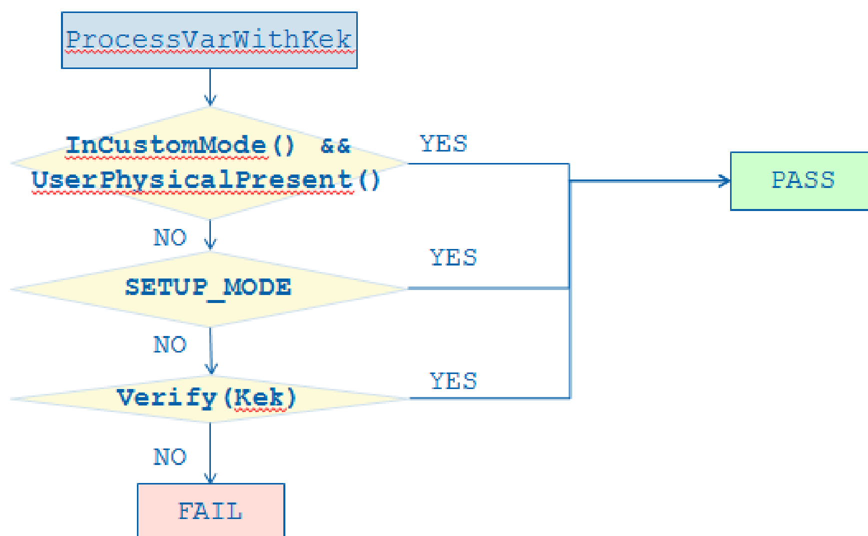

Figure 10: Variable Authentication Flow - ProcessVarWithKek

Authentication Flow - ProcessVariable

If system wants to update other authenticated variable, below flow is used.
First, if the variable requires PhysicalPresent, but system has no UserPhysicalPresent, update request is rejected. Second, if the variable is time based authenticated variable, authentication with time and creator's key is performed. Third, if the variable is count base authenticated variable, authentication with count and creator's key is performed. Last, if old variable has AUTH attribute, which does not patch current one, update request is rejected. Or it means no authentication is needed.

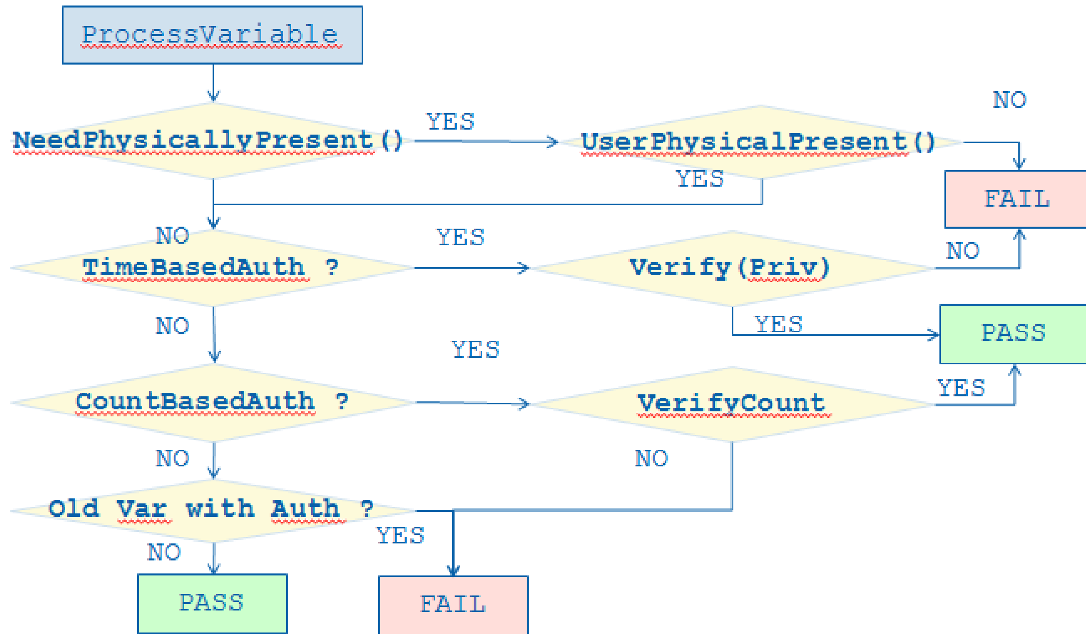

Figure 11: Variable Authentication Flow - ProcessVariable

Summary

This section describes the authentication part of SMM Authenticated Variable driver in EDKII, including why use authentication, authenticated variable input format, and authentication flow.

Part III - Variable

We have discussed SMM and authentication in previous chapters. Now let's focus on variable itself.

Variable Storage Format

First, how a variable is stored on a non-volatile storage, UEFI specification does not define the storage format. EDKII implementation chooses below format.

First variable exists in a special Variable Firmware Volume (FV). The variable FV can be identified by gEfiSystemNvDataFvGuid in File Syetem GUID field of FV header. The Authenticated Variable can be identified by gEfiAuthenticatedVariableGuid in GUID field of Variable Store Header. 0x5A in Format field of variable store header means this region is formatted. 0xFE in State field of variable store header means healthy.

Each individual variable is saved after variable storage header. 0x55AA in StartId of variable header means there is a new variable storage. 0x3F in State field means it is a valid variable added. 0x3D in state field means it is deleted. In some rare case, system reset during variable update, 0x7F in State field means only header is valid, no real variable data written. 0x3E in State field means this variable is delete transition.

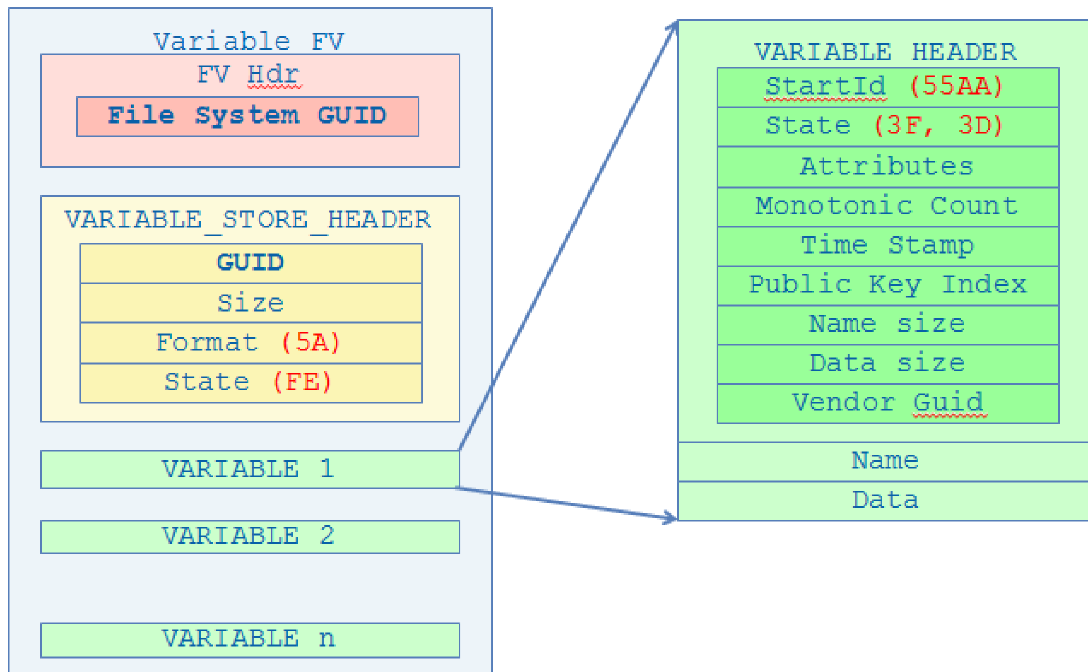

Figure 12: Variable Storage Format

Variable Update Flow

The State field is extremely useful on variable update. When system wants to update a variable, below flow is used.

1) The old variable state is marked as InDeleted.
2) New variable full header is added with state unchanged (0xFF).
3) New variable state is changed to Header Valid state.
4) New variable full data is added.
5) New variable state is changed to Added.
6) Old variable state is marked as deleted.

For a NOR flash, only one byte write from bit 1 to bit 0 can be atomic from hardware perspective. This State is designed to maintain atomicity of individual variable from software perspective. Each variable exists in ALL-or-NONE state. Partial variable is not allowed.

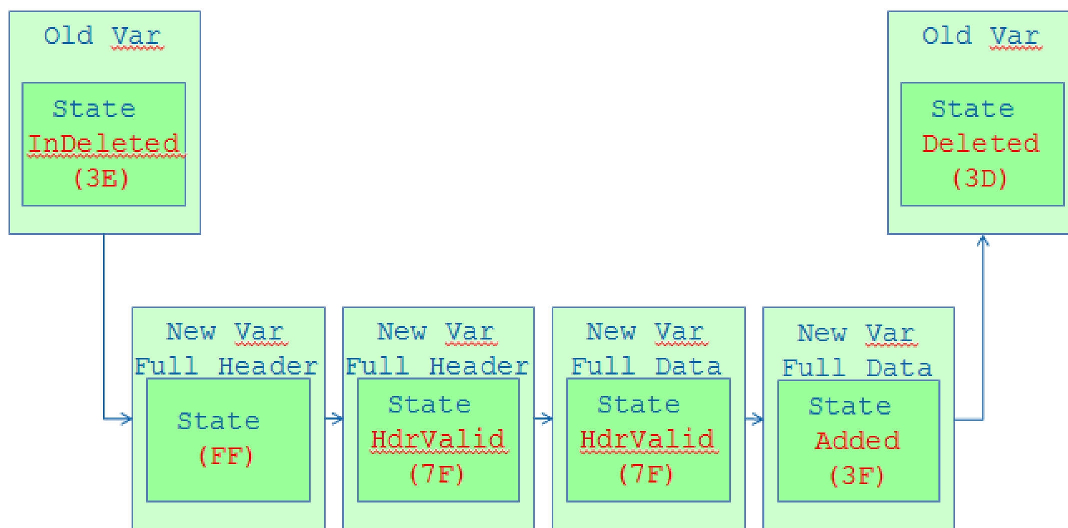

Figure 13: Variable Update flow

Variable Reclaim

From above variable update flow, when a variable is update. It is actually not UPDATED in original place, but marked as DELETED in original place, then ADDED in new place. If user updates variable several times, the non-volatile storage will have many DELETED variable, which is useless but occupies storage space.

In this case, there is a way called RECLAIM, to reorganize the variable region. It removes the DELETED variable to save space. Reclaim is triggered in below condition:

1) When update a variable or add a new variable, there is no enough free space.
2) On ready to boot event, there is no enough remaining free space.
3) On initialization, variable storage free space is not all 0xFF. (There must be something wrong.)

However, in EDKII reclaim is NOT supported during OS runtime, because it need erase entire flash block (NOR flash does not support write from bit 0 to bit 1), which is time consuming.

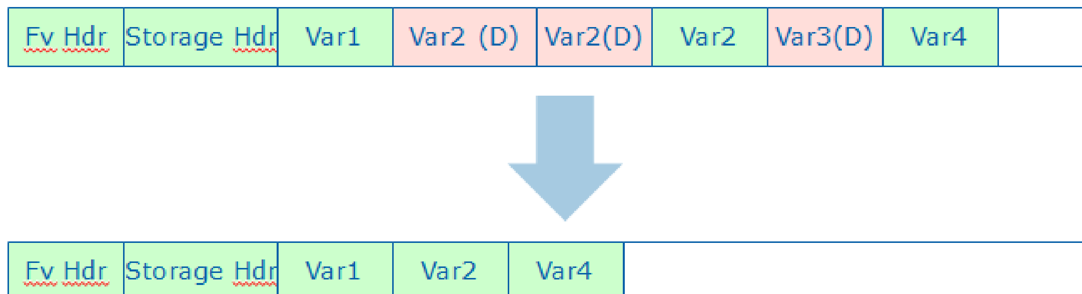

Figure 14: Variable Reclaim

Fault Tolerant Write

Individual variable atomicity is maintained by variable update flow. However, during variable reclaim, the flash block will be erased and be written again. In that period of time, if power lost or system shutdown due to user mistake, the variable firmware volume will be partially destroyed. That means if a variable crosses flash block, it might be partially correct. The atomicity is still broken. It is not acceptable.

Fault Tolerant Write (FTW) is designed to handle this situation. EDKII FTW driver is not included in the variable driver. It is a standalone driver to provide the capability of fault tolerant write. Every flash update driver can consume the FTW protocol API to update flash part in a safe manner.

Below is a high level picture of the fault tolerant write flash layout. A FTW driver requires 2 flash parts:

1) FTW working block. This is a block to record write action. It is the record block, not the data block.
2) FTW spare block. This is a real data block to save the data. It must be bigger than the size of block required to perform the update. In this case, it must be bigger than variable region.

In the FTW working block, FTW driver put a data structure to record the write request and write status. See below picture for detail.

The signature field of the WORKING_BLOCK_HEADER is used to identify if it is FTW working block. After the header, there will be multiple WriteQueueEntry's. Each WriteQueueEntry has one WRITE_HEADER and 1 or more WRITE_RECORD's. The number of WRITE_RECORD's is NumberOfWrites fields of the WRITE_HEADER. The most important fields are the Complete field of the WRITE_HEADER, the SpareComplete and the DestinationComplete of the WRITE_RECORD. Those fields record the status of write's and guarantee the fault tolerance.

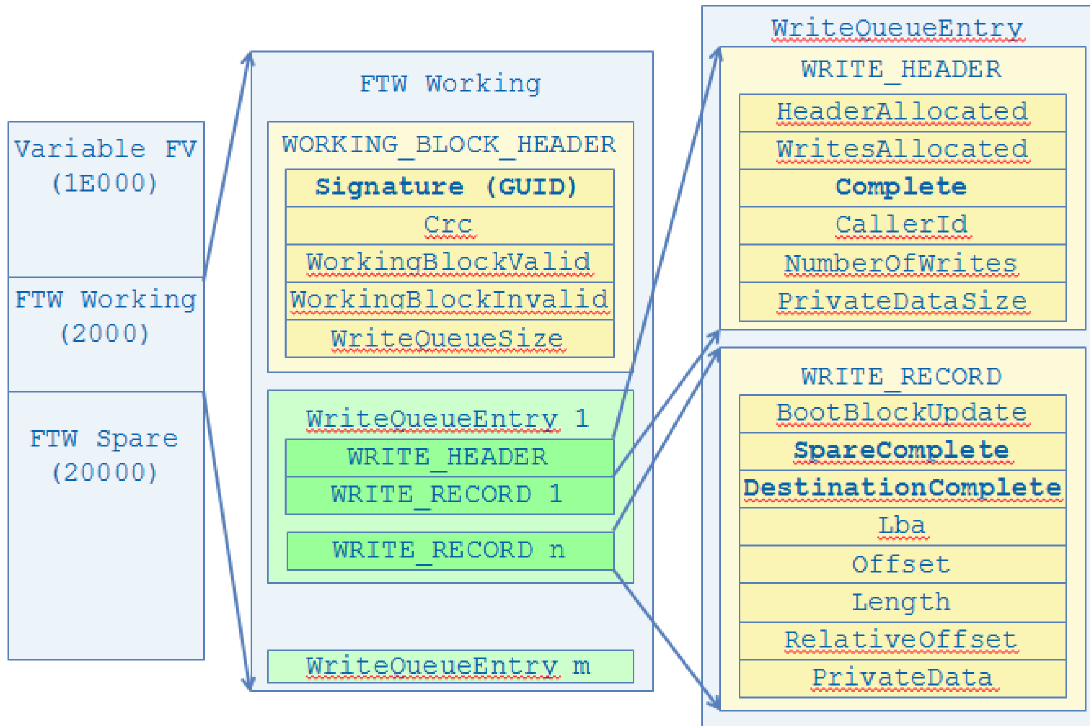

Figure 15: Fault Tolerant Write Flash Layout

Now, let's see how FTW->Write() works. There are a total of 8 steps involved.

1) Step 1, when FTW->Write() is invoked, this API will record the request in the FTW working block.
2) Step 2, this API finds SpareBuf on the FTW spare flash area and backs it up to memory.

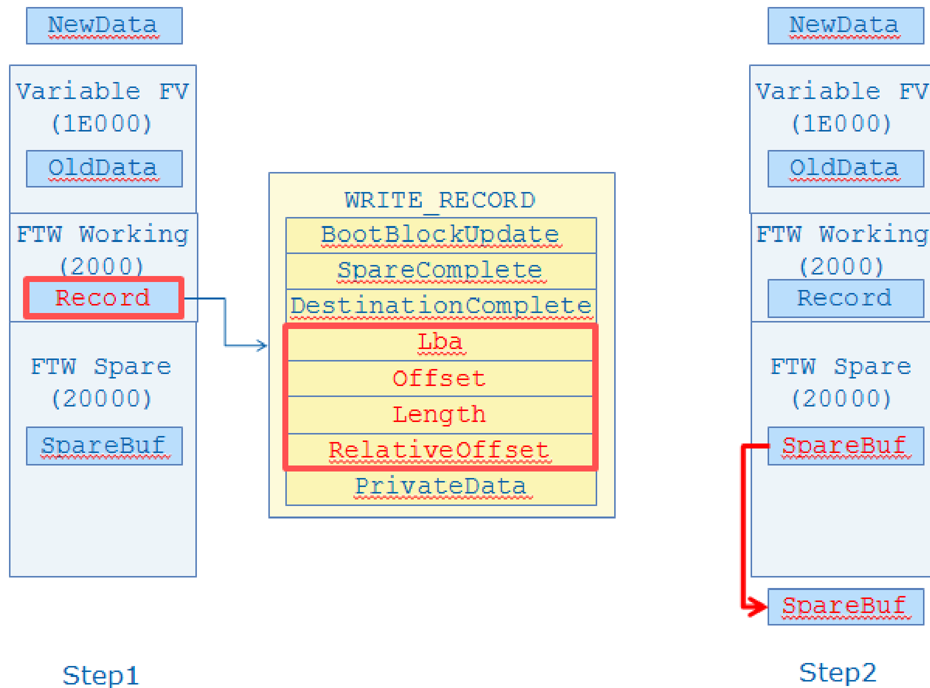

Figure 16: FTW write step 1 and 2

3) Step 3, this API writes NewData to the FTW spare block, instead of the Variable FV.
4) Step 4, after that, it sets the SpareComplete flag in the FTW working block.

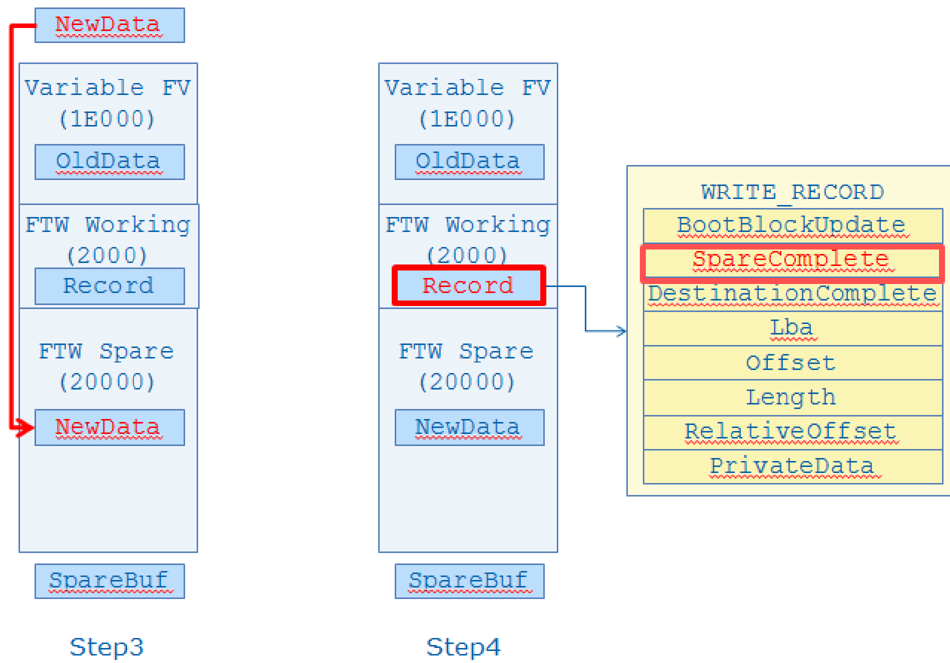

Figure 17: FTW write step 3 and 4

5) Step 5, this API writes NewData from FTW spare block to Variable FV.
6) Step 6, after that, it sets DestinationComplete flag in FTW working block.

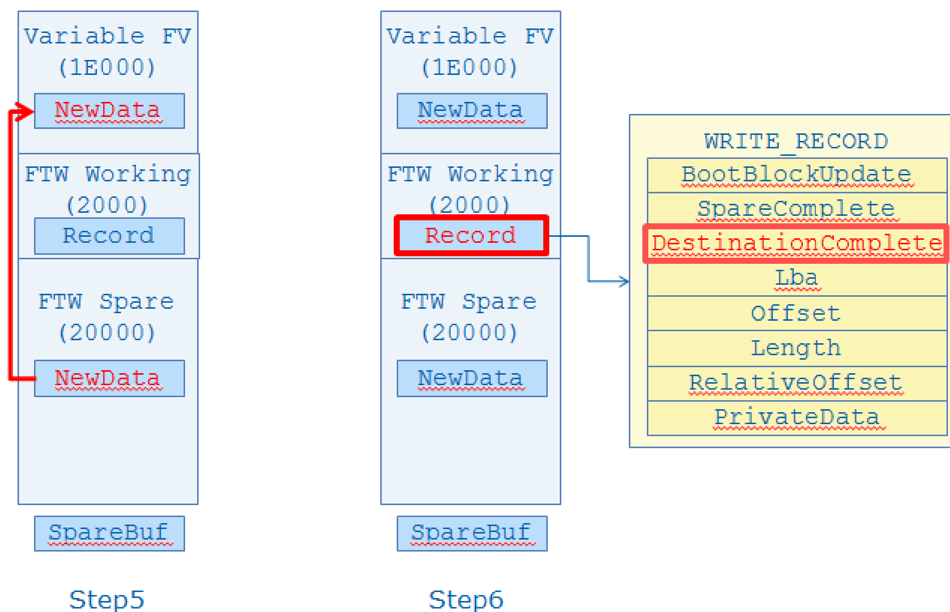

Figure 18: FTW write step 5 and 6

Step 5 is most important step, and we want to make sure it is fault tolerant. If system reset during step 5. Then SpareComplete flag is set, but DestinationComplete flag is not set. In next boot, the FTW driver will detect this situation and try to do recovery. The data is inside of FTW spare block, and all the LBA/size information is in FTW working block. So the corrupted variable region will be recovered in next boot.

7) Step 7, if it is last WRITE_RECORD associated with WRITE_HEADER, this API sets Complete flag in WRITE_HEADER.
8) Step 8, SpareBuf is restored in FTW space block. Then FTW->Write() finishes.

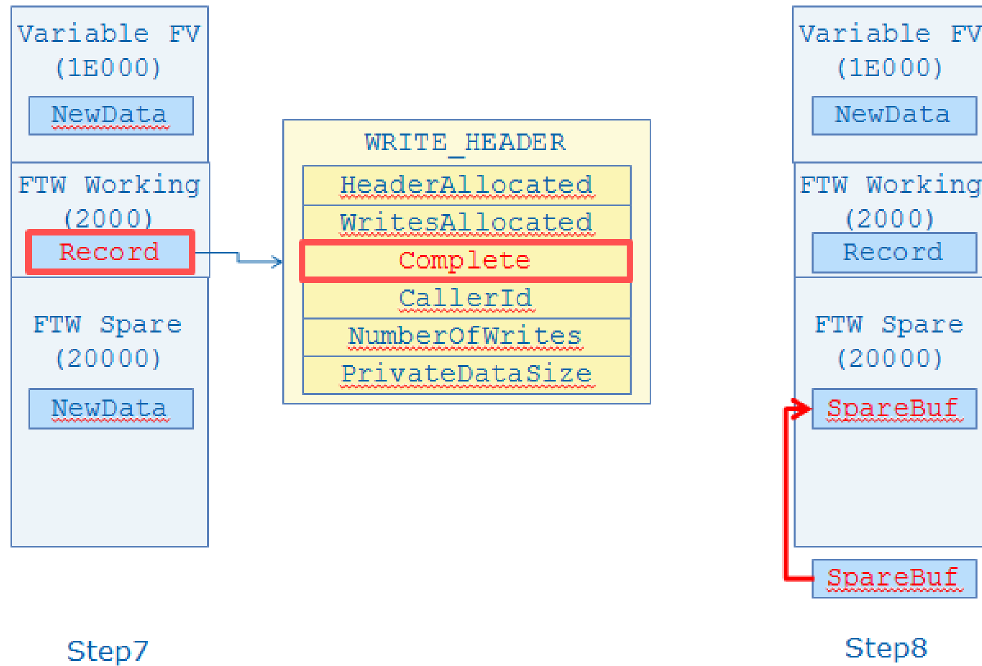

Figure 19: FTW write step 7 and 8

Summary

This section describes the variable part of SMM Authenticated Variable driver in EDKII, including variable storage format, variable update flow, and variable reclaim.

Part IV – Robust Consideration

Variable Lock

UEFI specification defines variable attributes could be non-volatile (NV), boot services access (BS), runtime access (RT), hardware error record (HR), count based authenticated write access, time based authenticated write access (AT), and append write. However, there is a need to support read only (RO) variable for some special usage.

EDKII implements EDKII_VARIABLE_LOCK_PROTOCOL, to support mark some variable to be READ ONLY, by adopt below rule:

1) Before EndOfDxe event, EDKII_VARIABLE_LOCK_PROTOCOL.RequestToLock() API is used to submit Variable Lock request.
2) This Lock request itself is volatile. That means user need call RequestToLock() in each boot.
3) After EndOfDxe event, RequestToLock() API is closed.

Variable lock policy rule is below:

1) Before EndOfDxe event, Variable Lock does not take effect.
2) After EndOfDxe event, Variable Lock takes effect in DXE or OS runtime phase. A locked Variable cannot be deleted or updated. A locked variable cannot be created, if it does not exist before.
3) Variable Lock does not take effect in SMM phase. Inside SMM, the variable can still be updated.

A full summary below:

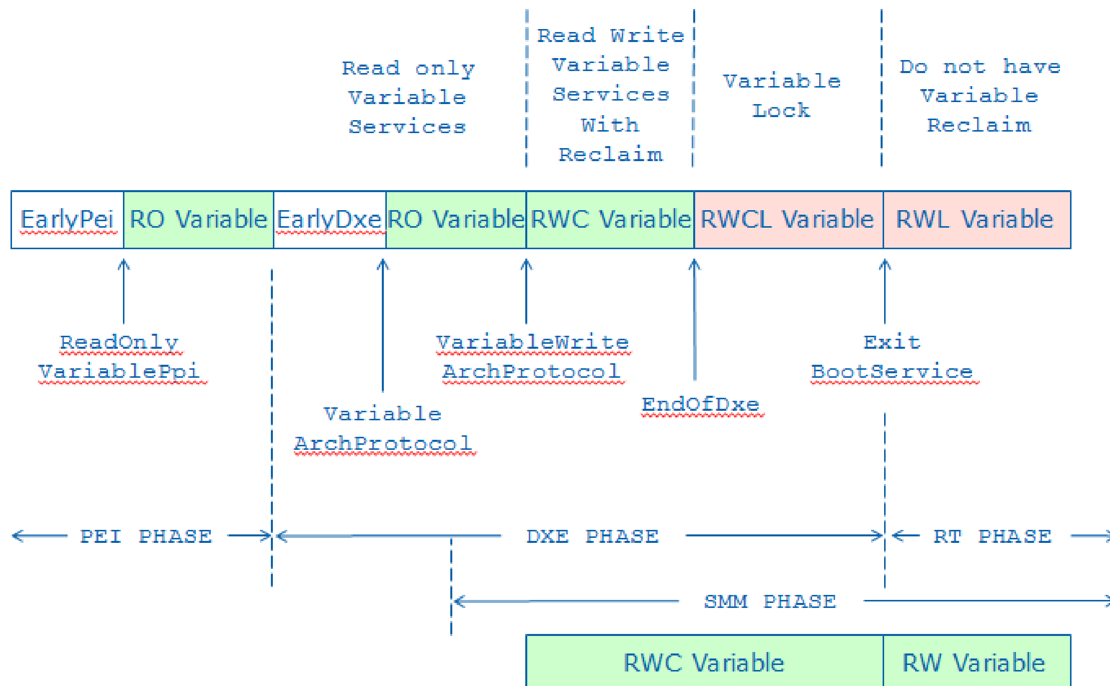

Figure 20: Variable Features in Each Phases

RO: means read only.
RW: means read write.
RWC: means read write, with reclaim feature.
RWL: means read write, with lock feature.
RWCL: means read, with reclaim and lock features.

Variable Check

A platform may define a setup variable used by UEFI Human Interface Infrastructure (HII), so that user may control platform setting by updating a setup user interface (UI). For example, user can configure a SATA controller to be in IDE mode or AHCI mode, or even RAID mode. A setup variable field SataMode is used. 1 means IDE mode, 5 means AHCI mode, 6 means RAID mode. Other values, such as 0, 2, 3, 4, or 7, are considered as invalid input.

The problem we were facing is that there is no way to check if some variable fields are valid and reject invalid variable update. In above example, when the code calls SetVariable() to update a setup variable, we hope variable driver can check one field of setup variable - SataMode to see if the setting is valid (1, 5, or 6), and reject the variable update if the setting is invalid.

In order to resolve above problem, we introduced "Variable Check" - the capability of checking variable content based on pre-defined policy – to EDKII variable driver.

EDKII Variable Checker only checks variable format in SetVariable() API. It does not checks variable format in GetVariable() API, because if all input variable is checked, it must be correct on get.

In order to manage all variable check NULL libraries and co-ordinate variable check handler registered by protocol, EDKII variable driver links a generic variable check manager. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Library/VarCheckLib.h and https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/VarCheckLib)

In EDKII, the variable check handler can be implemented as below 2 ways:

- A NULL library link with VarCheckLib. The library can be used if the handler itself is UEFI environment independent. The code can be reused in non-UEFI environment. For example:
  - The UEFI specification based variable check is at https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/VarCheckUefiLib
  - The UEFI HII specification based variable check is at https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/VarCheckHiiLib
  - The PI PCD based variable check is at https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/VarCheckPcdLib
- A standalone driver to consume protocol (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/VarCh

eck.h) produced by variable driver. The protocol can be used when the handler need complex logic to check variable or system state. Or the handler is platform specific, so that it might not be proper for variable driver binary distribution. For example:

- The TPM MOR Lock driver is at https://github.com/tianocore/edk2/tree/master/SecurityPkg/Tcg/MemoryOverwriteRequestControlLock

Below figure shows the whole relationship: Variable driver links VarCheckLib and produces VAR_CHECK_PROTOCOL. The implementation of VAR_CHECK_PROTOCOL is also in VarCheckLib. VarCheckHii, VarCheckPcd and VarCheckUefi are three default EDKII variable check handlers (SetVariableCheckHandlerHii/ SetVariableCheckHandlerPcd/ SetVariableCheckHandlerUefiDefined). They checked variable based on UEFI HII, PI PCD and UEFI specification. TPMMorLock is default variable check handler for TPM MOR variable (SetVariableCheckHandlerMor). The detail is defined in [Secure MOR].

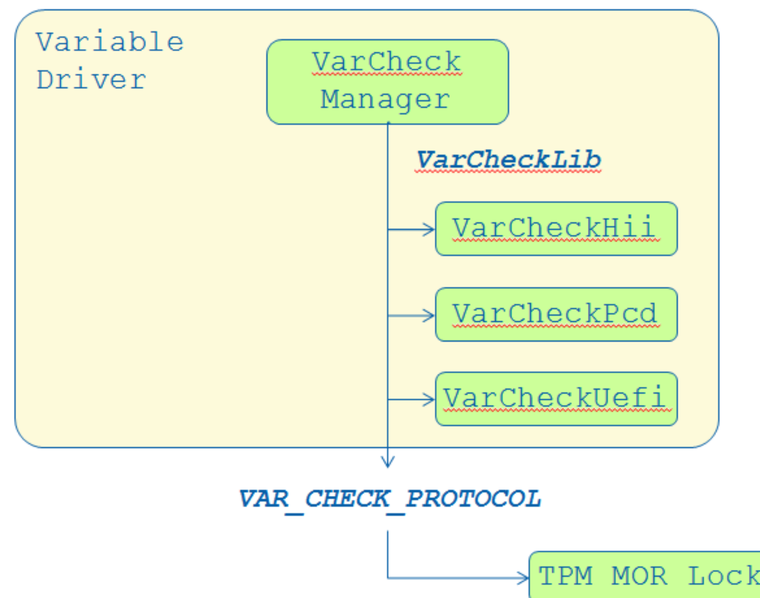

Figure 21: Variable Check

Finally, the variable driver has below layers:

1) Interface Layer: The API exposed to other driver. External interface is defined by UEFI specification as runtime variable services. Internal interface is defined by EDKII SMM variable driver itself, as software SMI, because EDKII variable driver need SMM as trusted execution environment. (This is introduced in Part I)
2) Software Logic Layer: The generic logic to do sanity check for variable. EDKII variable driver has 2 layer checks – a UEFI/PI/platform defined variable checker (This is being introduced in Part IV), and UEFI defined authentication. (This is introduced in Part II)
3) Hardware Storage Layer: Once variable passes sanity check, it will be written to storage area. The first sub-layer defines variable storage format, and the second sub-layer defines device access method. (These are introduced in Part III)

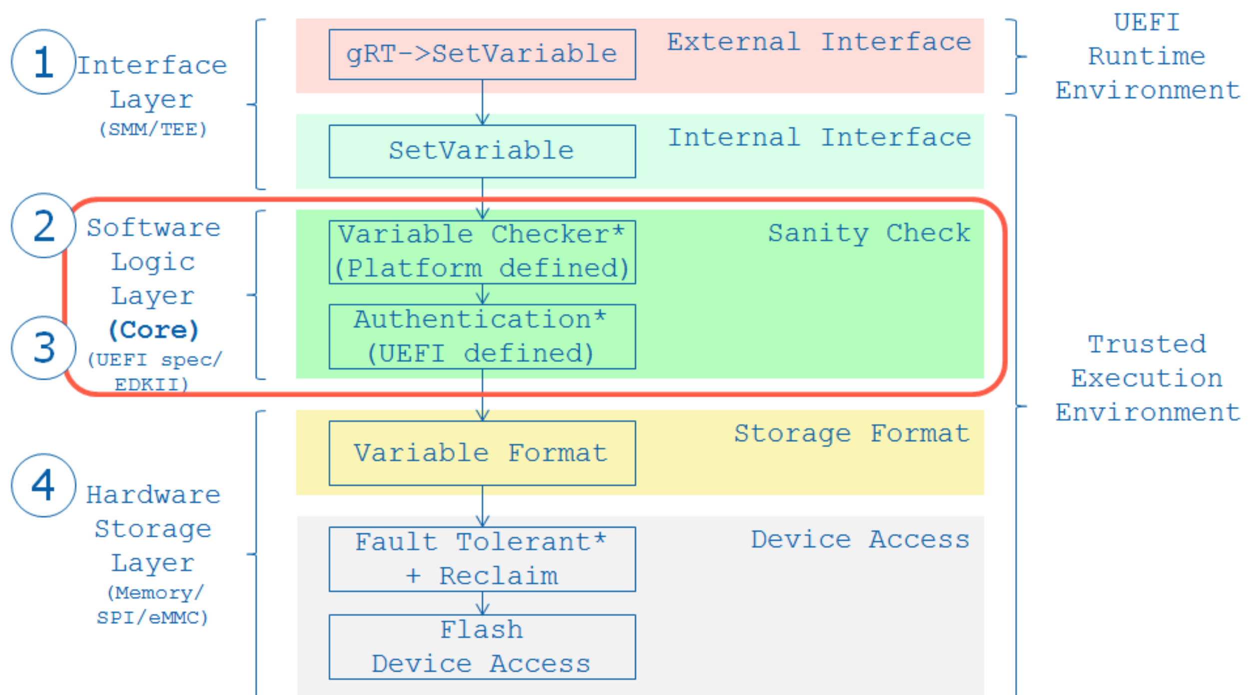

Figure 22: Variable Driver Design

In next 2 sections, we will introduce 2 variable check handlers as example, to show how checker works.

Variable Check – HII Based Checker

UEFI specification HII section defines some HII opcode can be linked with UEFI variable storage. (UEFI 2.5 specification, 31.2.5.6 storage). We can use data in HII IFR opcode to check the legal configuration of variable content. For example:

- **ONE_OF_OP**: variable data must be in OPTION list.
- **CHECKBOX_OP**: variable data must be between 0 or 1.
- **NUMERIC_OP**: variable data must be between MIN and MAX.
- **ORDERED_LIST_OP**: variable data must be an ordered list.

In order to achieve above check, VarCheckHii handler need get HII data from 2 sources:

- Static HII data: HII data built inside of firmware volume. EDKII build tool generate HII information to FFS raw section (See below figure). So that it is EDKII implementation specific way to get HII data. The typical usage is setup configuration data. The information can be got even if some special setup data is not installed to HII database due to current hardware configuration.

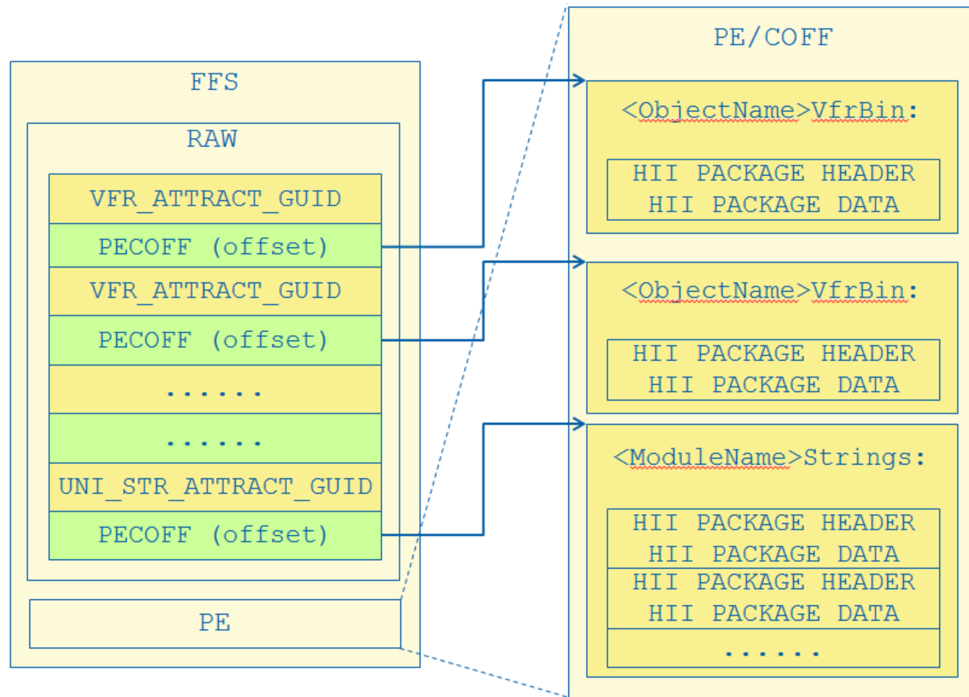

Figure 23: HII Build time info

- Dynamic HII data: HII data exposed by HII_DATABASE_PROTOCOL. This is UEFI defined way to get HII data from HII database. The typical usage is 3rd part option rom.

During initialization, VarCheckHiiGen() collects HII information from FV and HII_DATABASE_PROTOCOL and generate VarCheckHiiBin – a compact data structure to store ONE_OF, NUMERIC, CHECKBOX information (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/VarCheckHiiLib/InternalVarCheckStructure.h). During runtime, SetVariableCheckHandlerHii() refers VarCheckHiiBin to check if variable content is legal. If variable attributes are different, data size is different, or the content does not satisfy HII question, the variable content is treated as illegal, and EFI_SECURITY_VIOLATION is returned to reject the variable update.

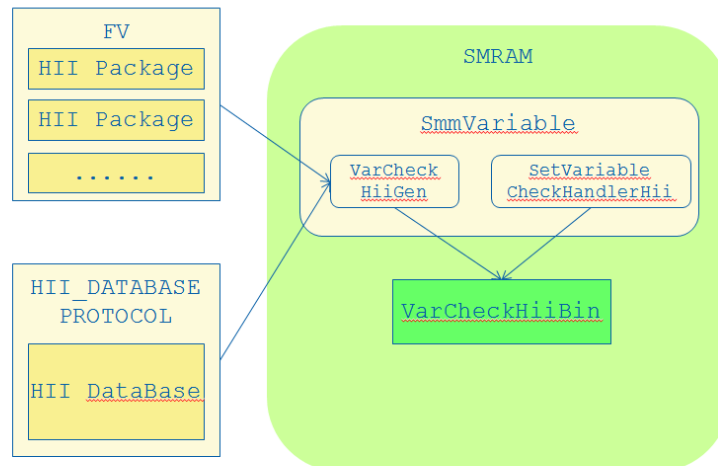

Figure 24: Variable Check HII

EDKII VarCheckHii handler just uses simple policy for ONE_OF, NUMERIC, CHECKBOX opcode, because they are most popular usage in platform setup variable. A known limitation is that HII inconsistent check is unsupported. The reason is that inconsistent check need expression parser support, it might need read other variable storage or buffer storage. The implementation might be too complicated.

Variable Check – PCD Based Checker

EDKII PCD is mapped to be a UEFI variable, if it is configured as PcdDynamicHii. EDKII PCD implementation also supports defining a set of valid configuration for a specific PCD in DEC file. For example:

- **@ValidList**: variable data must be in the list.
- **@ValidRange**: variable data must be in the range.

So that we have a way to check if PCD mapped variable is legal.

The EDKII build tool generates the information so that platform may put binary data to FFS raw section. (The example build rule and definition is at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/VarCheckPcdLib/VarCheckPcdLib.inf.) The binary format can be found at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/VarCheckPcdLib/VarCheckPcdStructure.h. During initialization, LocateVarCheckPcdBin () uses EDKII implementation way to get VarCheckPcdBin from FFS raw section. During runtime, SetVariableCheckHandlerPcd() refers VarCheckPcdBin to check if variable content is legal. If variable attributes are different, variable size is too small, or the content does not satisfy PCD valid list or valid range, the variable content is treated as illegal, and EFI_SECURITY_VIOLATION is returned to reject the variable update.

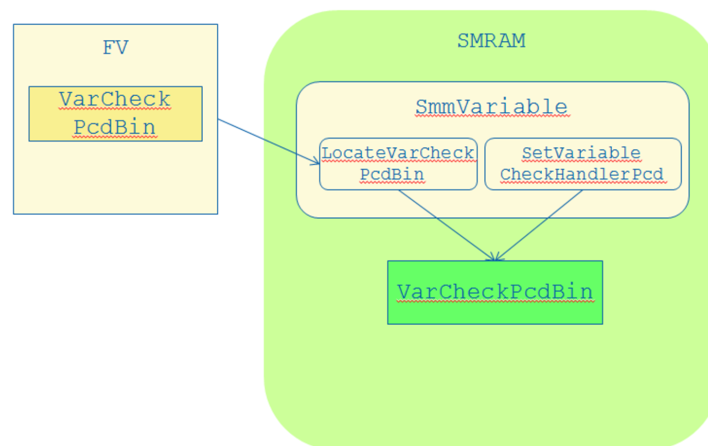

Figure 25: Variable Check Pcd

Because PCD may only include variable offset, but not variable size, it is impossible to know variable size. We can only check if variable is too small, but we cannot know if variable is too big. This is considered as known limitation of VarCheckPcd.

EDKII VarCheckPcd handler just uses simple policy for @ValidList and @ValidRange. A known limitation is that PCD @Expression check is unsupported. The reason is that @Expression check need expression parser support. The implementation might be too complicated.

Variable Check – Variable Property

In order to check variable, a driver can call `RegisterSetVariableCheckHandler()` to register SetVariable check handler. If the policy is simple, a driver can just call `VariablePropertySet()` to define "Property" for a specific variable.

Current variable property structure has below fields:

- Attributes: UEFI defined variable attributes (BS/RT/NV/...)
- Property: non-UEFI defined variable attributes (READ_ONLY)
- MinSize: The minimal size of variable
- MaxSize: The maximal size of variable

A platform may set policy for each known variable. For example:

```
========================
typedef struct {
  EFI_GUID                      *Guid;
  CHAR16                        *Name;
  VAR_CHECK_VARIABLE_PROPERTY   VariableProperty;
} INTERNAL_VARIABLE_ENTRY;

INTERNAL_VARIABLE_ENTRY mCoreVariableList[] = {
  {
    &gEfiMemoryTypeInformationGuid,
    EFI_MEMORY_TYPE_INFORMATION_VARIABLE_NAME,
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS,
      sizeof (EFI_MEMORY_TYPE_INFORMATION),
      sizeof (EFI_MEMORY_TYPE_INFORMATION) * (EfiMaxMemoryType + 1)
    },
  },
  {
    &gEfiFirmwarePerformanceGuid,
    EFI_FIRMWARE_PERFORMANCE_VARIABLE_NAME,
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS,
      sizeof (FIRMWARE_PERFORMANCE_VARIABLE),
      sizeof (FIRMWARE_PERFORMANCE_VARIABLE)
    },
  },
  {
    &gMtcVendorGuid,
    MTC_VARIABLE_NAME,
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
```

```
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
      sizeof (UINT32),
      sizeof (UINT32),
    },
  },
  {
    &mPcAtRtcVariableGuid,
    L"RTC",
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
      sizeof (UINT32),
      sizeof (UINT32),
    },
  },
  {
    &mPcAtRtcVariableGuid,
    L"RTCALARM",
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
      sizeof (EFI_TIME),
      sizeof (EFI_TIME),
    },
  },
  {
    &gEfiGenericVariableGuid,
    DEBUG_MASK_VARIABLE_NAME,
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS,
      sizeof (UINTN),
      sizeof (UINTN),
    },
  },
};
=======================
```

Variable Quota Management

Any system resource is limited, including variable storage. A typical platform may allocate around 128K~256K region for variable storage. Sometimes, variable region must be full and SetVariable() returns OUT_OF_RESOURCE. It might happen when a QA engineer runs test in UEFI shell, or a hacker tries to attack system in OS. How we handle that?

Most modern OSes (Linux and Windows) have disk quota management to set a limit for disk storage for special user or group, give notification if disk is near full, and let user clean up. In EDKII, we enabled similar way for variable quota management.

Variable Quota Management – Allocation

There is a set of variable size related PCDs defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/MdeModulePkg.dec .

- PcdFlashNvStorageVariableSize: The whole variable storage region size on flash.
- PcdMaxVariableSize: The maximum size of a single non-HwErr type variable.
- PcdMaxAuthVariableSize: The maximum size of a single authenticated variable.

- PcdMaxHardwareErrorVariableSize: The maximum size of single hardware error record variable.
- PcdVariableStoreSize: The size of volatile buffer.
- **PcdHwErrStorageSize**: The size of reserved HwErr variable space
- **PcdBoottimeReservedNvVariableSpaceSize**: The size of NV variable space reserved at UEFI boot time.
- **PcdMaxUserNvVariableSpaceSize**: The size of maximum user NV variable space.

The last three PCDs are used for Quota management. **PcdHwErrStorageSize** indicates the space reserved for UEFI hardware error logging variable only.
PcdBoottimeReservedNvVariableSpaceSize indicates the space reserved for UEFI boot time. Even if malicious code writes variable space till out of resource state at OS runtime, BIOS can still write new variable at boot time.

PcdMaxUserNvVariableSpaceSize indicates the maximum space can be used for user NV variable. EDKII Variable driver divides variable into 2 groups: System Variable and User Variable. Below types of variables will be regarded as System Variable after EndOfDxe:

- **UEFI defined variables** (gEfiGlobalVariableGuid and gEfiImageSecurityDatabaseGuid variables at least). For example: L"ConIn", L"ConOut", L"db", L"dbx".
- **Variables managed by Variable driver internally**. For example: L"CustomMode", L"VendorKeysNv".
- **Variables need to be locked**, they MUST be set by VariableLock protocol. For example: L"PhysicalPresenceFlags".
- **Important variables during platform boot**, their property SHOULD be set by VarCheck protocol. For example: L"MemoryOverwriteRequestControl", L"MemoryOverwriteRequestControlLock", platform setup variable for system configuration.

If a variable is not system variable, it is a user variable.
NOTE: System Variable can be authenticated variable or non-auth variable. User variable can also be authenticated variable or non-auth variable.

The reason EDKII variable driver supports limit for user NV variable is that variable driver wants to make sure there is enough space for system variable. System variable is critical for system boot. User variable is less important.

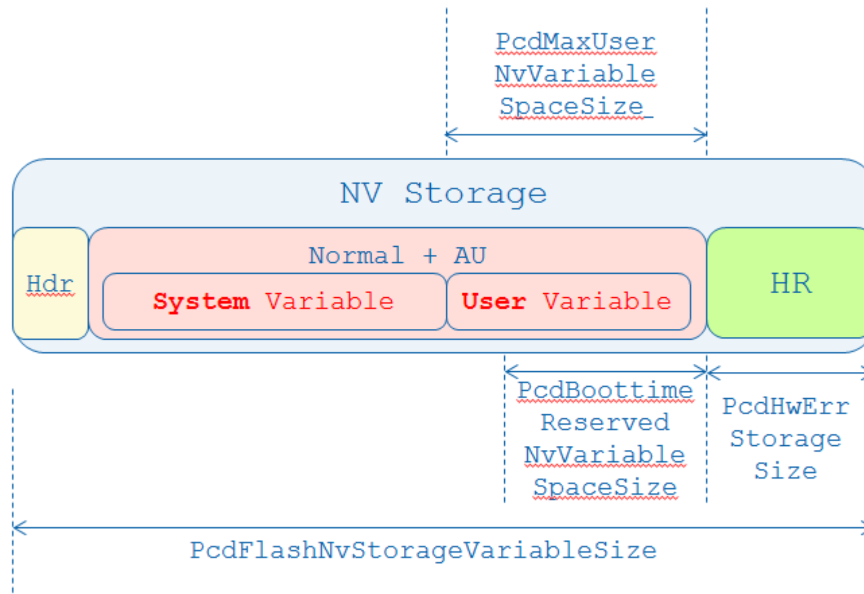

Figure 25: Variable Quota Allocation

Variable Quota Management – Error Logging

When quota limitation is hit, EDKII variable driver need record the error, so that a platform owner may take action to recover system to good state.

This error information is recorded in L"VarErrorFlag" variable (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/VarErrorFlag.h). EDKII variable driver uses one byte to record the error. 0xFF means no error. 0xEF means system variable out of space. 0xFE means user variable out of space. If system gets 0xEE later, it means both system variable and user variable are out of space.

Variable Quota Management – Recovery

In next boot, if a platform detects L"VarErrorFlag" in error state, it may use platform specific way to clean up some unused variables. The system variable can be handled differently with user variable. The possible implementation could be:

- System variable out of space: Enter Setup to let user load default setting; or force clear variable region and treat as first boot.
- User variable out of space: Prompt a setup page and list all user variables to let user select which user variable to be deleted; or just delete all user variables(https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Library/PlatformVarCleanupLib).

Summary

This section describes the variable robust consideration of SMM Authenticated Variable driver in EDKII, including variable lock, variable check, and variable quota management.

Conclusion

SMM Authenticated Variable is an important component in UEFI secure boot. This paper describes detail work flow and data structure of SMM Authenticated Variable driver in the MdeModulePkg and SecurityPkg.

Glossary

db – Authorized Image Signature Database, see UEFI specification, secure boot section.

dbt - Authorized Timestamp Signature Database, see UEFI specification, secure boot section.

dbr - Recovery Signature Database, see UEFI specification, secure boot section.

dbx – Forbidden Image Signature Database, see UEFI specification, secure boot section.

FTW – Fault Tolerant Write. A robust way to update variable.

KEK – Key Exchange Keys, see UEFI specification, secure boot section.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

PK – Platform Key, see UEFI specification, secure boot section.

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime

TEE – Trusted Execution Environment.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Predominate interfaces are in the boot services (BS) or pre-OS. Few runtime (RT) services.

References

[SECURE1] Jacobs, Zimmer, "Open Platforms and the impacts of security technologies, initiatives, and deployment practices," Intel/Cisco whitepaper, December 2012, http://uefidk.intel.com/sites/default/files/resources/Platform_Security_Review_Intel_Cisco_White_Paper.pdf

[SECURE2] Magnus Nystrom, Martin Nicholes, Vincent Zimmer, "UEFI Networking and Pre-OS Security," in *Intel Technology Journal - UEFI Today: Boostrapping the Continuum*, Volume 15, Issue 1, pp. 80-101, October 2011, ISBN 978-1-934053-43-0, ISSN 1535-864X https://www.researchgate.net/publication/235258577_UEFI_Networking_and_Pre-OS_Security/file/9fcfd510b3ff7138f4.pdf

[SECURE3] Zimmer, Shiva Dasari (IBM), Sean Brogan (IBM), "Trusted Platforms: UEFI, PI, and TCG-based firmware," Intel/IBM whitepaper, September 2009, http://www.cs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/SF09_EFIS001_UEFI_PI_TCG_White_Paper.pdf

[Secure MOR] Microsoft Secure MOR implementation. https://msdn.microsoft.com/en-us/library/windows/hardware/mt270973(v=vs.85).aspx

[EDK2] UEFI Developer Kit www.tianocore.org

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

[UEFI Book] Zimmer,, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.3 www.uefi.org

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDKII BIOS architect, EDKII FSP package maintainer, EDKII TPM2 module maintainer, EDKII ACPI S3 module maintainer, with Software and Services Group at Intel Corporation. Jiewen is member of UEFI Security Sub-team and PI Security Sub-team in the UEFI Forum.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group at Intel Corporation. Vincent chairs the UEFI Security and Network Sub-teams in the UEFI Forum and has been working on the EFI team at Intel since 1999.

Star Zeng (star.zeng@intel.com) is EDKII BIOS engineer with Software and Services Group at Intel Corporation.

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel, the Intel logo, Intel. leap ahead. and Intel. Leap ahead. logo, and other Intel product name are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright 2015 by Intel Corporation. All rights reserved